Design Patterns Induction from Multiple Versions of Software

D Janakiram and J Rajesh
Distributed and Object Systems Lab
Department of Computer Science & Engg.,
Indian Institute Of Technology Madras, India.
{djram, rajesh}@cs.iitm.ernet.in

Abstract. *Refactoring using design patterns yields high quality software systems. Automatically inferring patterns from code for refactoring reduces errors and minimizes cost. We have developed a tool, JIAD (Java based Intent-Aspects Detector) which infers patterns from a single version of code. This tool attempts to infer 11 GOF design patterns from the code wherever applicable. However, patterns which are mainly meant for extendibility, may not be accurately inferred using this tool. In order to infer such patterns, we propose an approach which uses the knowledge of multiple versions of the software. This approach is an extension to our previous work, JIAD. Thus, by having the knowledge from multiple versions of the software, the induction*[1] *can be made more accurate. Induction from multiple versions ensures that the introduced patterns in the code improve the quality and reduce maintenance effort. This paper explains the approach with an example of Visitor pattern , issues and assumptions.*

Keywords: *Design Patterns, Versions, Refactoring, Maintainability, Prolog, TyRuBa, Intent-Aspects*

1 Introduction

Refactoring is "the process of improving the quality of the software systems without changing the external behavior of the system" [1]. Refactoring has gained increased attention in the object-oriented community. Design patterns play a key role in developing object-oriented software systems. Design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [2]. The software systems developed using design patterns are more reusable and extendable. Hence, refactoring using design patterns leads to high quality software.

There are tools [3][4][5][6] to perform behavior preserving transformations using design patterns. The inputs to these tools are the applications to be refactored, design patterns and portions of the code that should be refactored using

[1] Henceforth, in this paper "Inference" and "Induction" are used interchangeably

the selected design pattern. The portion of the code and the design pattern are manually identified from the application. This process is tedious, error prone and costs more. We have developed a tool, JIAD[7], which automatically infers patterns and the region of the code to which they are applied from single version of software. However, some of the patterns which are mainly meant for improving extendibility may not be accurately inferred from this tool. Capturing semantics of extendibility from a single version of the software is very difficult and sometimes impossible. Thus, inferring these kinds of patterns from the code requires some additional knowledge about future extensions. This knowledge can be easily acquired if there are multiple versions of the software.

This paper explains the approach for inferring patterns from multiple versions of a software with an illustrative example. In this approach, the knowledge about each version of the software is represented in the form of Prolog [8] facts. This is used to reason about the code. Rules for inferring Intent-Aspects(IAs) [9] corresponding to each pattern are written in Prolog. Two versions of the software are given as input to the tool and it verifies the rules for each pattern. The IAs and the corresponding pattern are produced as output if the proper rules are satisfied. The approach is an extension to JIAD. The approach in this paper is restricted to two versions. This paper also gives some of the issues and assumptions made while implementing the inference rules.

In this paper, section 2 presents the motivation for induction of patterns from multiple versions. Section 3 gives a brief overview of the proposed approach of the tool and presents few assumptions that are made while inducting patterns. Section 4 presents the rules for detecting IAs of Singleton, Bridge, Observer and Visitor patterns. Section 5 illustrates the approach with an example of Visitor pattern and it also presents the Prolog rules for inferring IAs of Visitor instance. Section 6 describes the related work. Finally, section 7 presents some conclusions and future work related to inference of patterns.

2 Motivation

To the best of our knowledge, there are no design pattern inference tools which refactor code/design. We have developed a tool, JIAD, to infer some of the GOF patterns. This tool infers patterns based on a single version of the software system. Using this tool some of the patterns can be accurately inferred with single version alone. However, there are patterns such as : Bridge, Visitor, Observer, Chain of Responsibility, Composite, etc which are mainly meant for providing extendibility to the software system. Some of these patterns can be inferred using JIAD. However, the inference may not be accurate[2]. Since it is very difficult to capture semantics of extendibility without having additional information (later versions design/code knowledge), it is not possible to accurately infer them from a single version. If the inference is not accurate, it leads to more complex systems.

[2] Henceforth, inference accuracy refers to 70-80%. It is not possible to make it cent percent accurate

Using the knowledge from multiple versions, it is possible to accurately infer patterns.

3 The Approach

As mentioned earlier in this paper, this approach is an extension to JIAD. In this approach, JIAD takes two versions of the same software as input. *ParserAndFactsGenerator* component of JIAD takes the input and parses the source code. It generates Prolog facts corresponding to each version of the software and stores them in a temporary location. Generated Prolog facts have the knowledge about the application code. The inference rules to detect IAs for each pattern are represented in the form of Prolog facts and they are stored in a temporary file called rule-base. The rules for inferring patterns from multiple versions are different from those for a single version. Since the inference here mainly concentrates on patterns which are mainly meant for extensions, the rules are formed by keeping in view the major changes to later versions of the software. Prolog interpreter takes the rule-base and fires them against the set of facts from the two versions. It outputs the IAs and its corresponding patterns once their rules are satisfied.

3.1 Assumptions

The rules are formed based on the following assumptions:

1. Changes from the first version to the next version are treated as major changes. The major changes are changes which introduce additional classes, methods, etc. This is because, the patterns to be inferred here are meant for extendibility. In order to capture semantics of extendibility, it is necessary to have these extensions. It might be possible to have different versions with small modifications (changing some of the class, method and field names, inserting some new statements in a method, adding additional parameters, etc) to previous versions. These are treated as minor changes.
2. It is possible to have different versions by just changing class, method and attribute names. To make the rules simple and easy to implement, it is assumed that names of the classes, methods and attributes are the same in the later version.

4 Rules for Inferring Design Patterns

This section presents informal and formal (Prolog) rules for inferring IAs and their corresponding patterns. These rules are formed after analyzing applicability, structure and consequence sections of design patterns extensively, by using the Kerievsky [10] catalog and some of the metrics proposed in [11]. Different versions of the same software are observed while forming rules to identify IAs.

1. **Singleton**
   This pattern is applicable if any of the following rules are satisfied.

- There exists only a single instance for a class in both versions of the software.
- There exist global variables in a version.

2. **Bridge**
   This pattern is applicable if all of the following rules are satisfied.
   - There exists an inheritance hierarchy, say X_1, in the initial version.
   - There exists an overridden method (overridden in all the derived classes) in the hierarchy such that it calls other methods.
   - Each overridden method is in isomorphism [12](i.e. number of methods called is same across all derived classes and there exists a mapping from each method called in one derived class to each method called in the other derived class) to the same overridden method in all derived classes.
   - There exists an inheritance hierarchy, say X_2, in the later version which is a super set of hierarchy with rules 1,2 and 3.
   - There exists a derived class in X_2 which is an abstract class, with a method which calls the overridden method. This subtree should also satisfy rules 2 and 3.
3. **Observer**
   This pattern is applicable if all of the following rules are satisfied.
   - There exists a class X, which has references to a set of at least 2 other concrete classes, say ObsSet1 in the initial version.
   - X has at least two operations which are used to set and get the updated values from the classes of ObsSet1.
   - X has a notify operation which updates all of the N other objects through a common method of ObsSet1.
   - For each of the N classes, it has a reference to class X and it calls the get and set operations of the class X.
   - There exists a set of classes, say ObsSet2, which are referenced from same X in the later version.
   - There is an abstraction among the classes in the ObsSet1.
   - ObsSet1 is a subset of ObsSet2.
   - All the properties which are applicable to ObsSet1 are also applicable to ObsSet2.
4. **Visitor**
   This pattern is applicable if all of the following rules are satisfied.
   - There exist inheritance hierarchies such that base class B, has references to set of classes say X, in both versions.
   - There exist overridden methods which call the methods of X.
   - Each of the overridden method calls at least one method of a class from X.
   - Number of overridden methods in each class is equal to the number of classes referenced.
   - There exists the same inheritance hierarchy with base class B in the later version as in the initial version. That is the hierarchy does not change.
   - The set of classes referenced from B in the later version is a superset of the set of classes referenced from B in the initial version.

– The set of overridden methods in each of the classes in the hierarchy in the later version is superset of the set of overridden methods in the corresponding classes in the initial version.

Once the rules corresponding to a pattern are satisfied, the IAs corresponding to those rules and its pattern are given as output from the tool. These IAs correspond to the later version of the software. IAs of version 2 are a superset of version 1.

5 An Illustrative Example: Visitor Pattern

This section explains the approach with an example. Visitor pattern is inferred from two versions of the software using the Prolog rules, shown in Table 1. The designs corresponding to these versions are shown in Figures 1 and 2 respectively. For the sake of clarity, other parts of the design structures are not included in the Figures 1 and 2. The design shown in Figure 1 is not easily extendible because each time we want to introduce a new functionality, the whole hierarchy should be altered. Since it is difficult to say that the **Document** hierarchy is fixed or new functionalities will be added from the knowledge of a single version, it is difficult to judge the applicability of Visitor pattern just by looking at single version. By using version 2 shown in Figure 2, information necessary to infer the IAs corresponding to Visitor pattern is acquired.

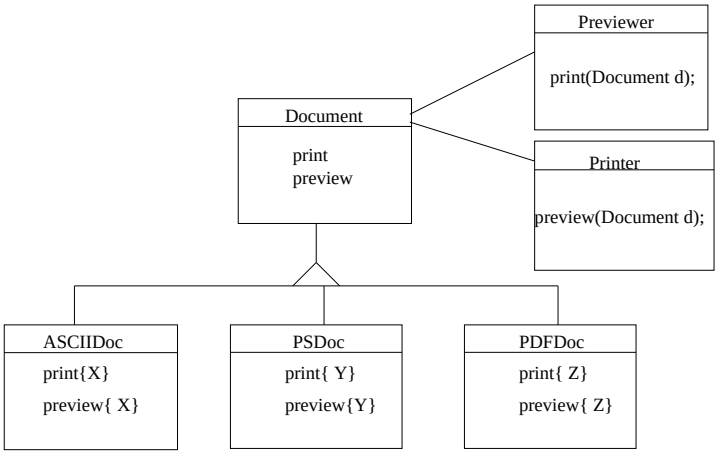

Fig. 1. Low Quality Design for Document Editor in Version 1

Assume that version 1 of the software whose code contains the structure of the design shown in Figure 1 is stored in the directory ***/homespace/parijatham/rajesh/Version1***. Similarly, version 2 of the software is stored in ***/homespace/parijatham/rajesh/Version2*** with the design as shown in Figure 2. The source

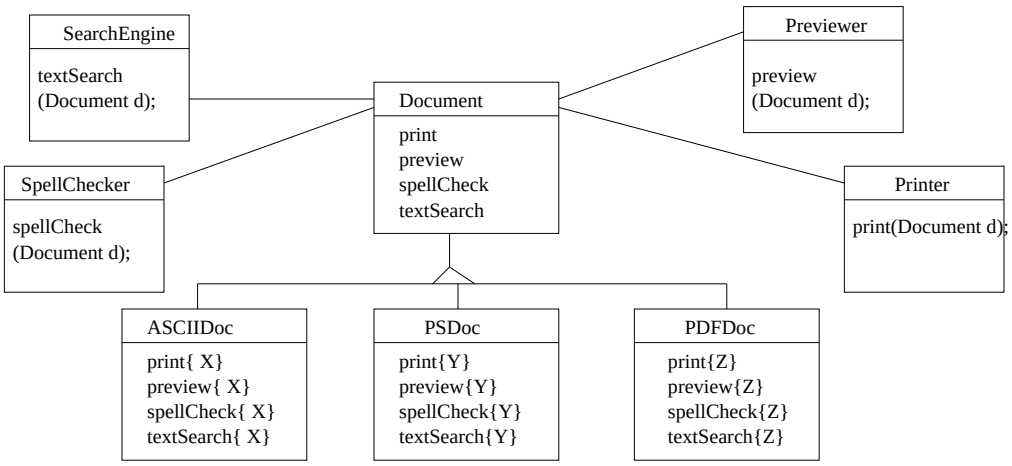

Fig. 2. Low Quality Design for Document Editor in Version 2

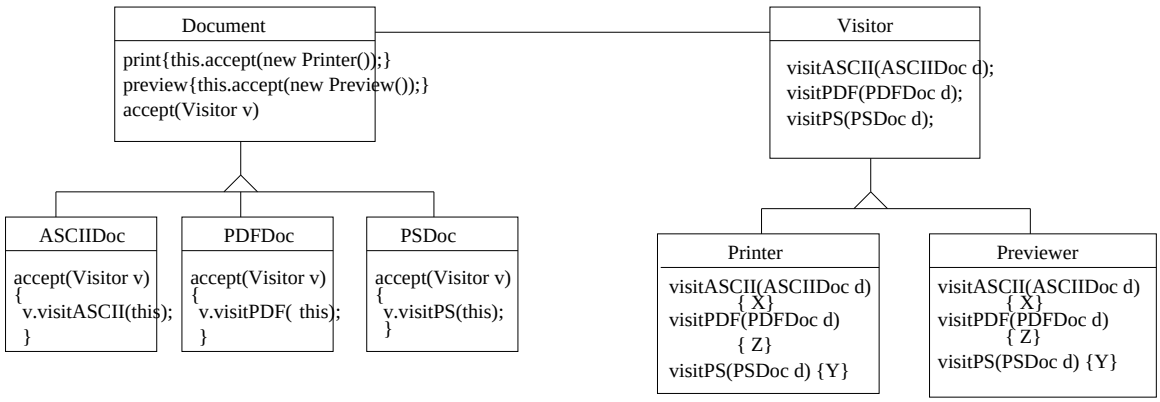

Fig. 3. Visitor Pattern Applied to Design in Version 1

code related to both of these versions is represented in the form of Prolog facts. The Prolog rules corresponding to IAs of Visitor pattern are matched against the facts corresponding to both versions. Versions are differentiated by using the path of each version. Since each application has only one main method, it is possible to get the path where this main method(file which has main method) is stored.

Primary rule for visitor pattern is **visitor_pattern**. This rule first gets the path information for both versions into *ListofVersions* list variable. The variables *V1* and *V2* have path of version 1 and 2 respectively. For the example shown in Figure 1 and 2, the values associated with *V1* and *V2* are :
*V1=**/homespace/parijatham/rajesh/Version1***
*V2=**/homespace/parijatham/rajesh/Version2***
object_structure sub rule is called for each version with the version path. For version 1, this rule first searches for inheritance hierarchies, in this case, variable *Hier1* has [**'ASCIIDoc', 'Document', 'PDFDoc', 'PSDoc'**] classes associated with it. **Document** class is the base class for this hierarchy. Next, it checks for references to element classes from the base class. It gathers all element classes which are associated with it. In this case, [**'Printer','Previewer'**] are two such element classes (*ListofElementClasses*). Next, this rule gets all the method names in the base class. Variable *LM* has values [**'print','preview'**].

Next, it checks for another sub rule, **called_methods**. This rule finds all the overridden methods in the derived classes. Next, the rule finds the list of methods called from the overridden methods of the derived classes. Then it checks whether the called methods belong to the classes in the *ListofElementClasses or EleList1*. It also checks if there exist same overridden methods across all derived classes and if the list of methods called from each of these overridden methods in all the derived classes are same. Called method list in this case is *CalledMethodList=*[**'Printer.print()', 'Previewer.preview()'**]. This is associated with *CalledMethods1* in **object_structure**.

Similarly, for version 2, **object_structure** and **called_methods** rules are verified with the version 2 path. The information associated with version 2 are as follows:
Hier2=[**'ASCIIDoc', 'Document', 'PDFDoc', 'PSDoc'**]
EleList2=[**'Printer', 'Previewer', 'SpellChecker', 'SearchEngine'**]
CalledMethods2= [**'Printer.print()', 'Previewer.preview()', 'SpellChecker .spellCheck()', 'SearchEngine.textSearch()'**]

Next, it checks whether the hierarchy in version 1 is fixed (it is not going to change in later versions). This is verified by comparing the two hierarchies in both versions. In this case *Hier1* and *Hier2* are the same. Next, it checks for extensions in the later versions with respect to functionalities. In this case, *CalledMethods1* is a subset of *CalledMethods2*. It means that new features are added in version 2. Similarly, it checks for the classes which implement these new features. That is, *EleList1* is a subset of *EleList2*.

Thus, it satisfies all the rules of IAs of Visitor pattern mentioned in section 4. It returns the IAs corresponds to the Visitor pattern. The IAs, in this

Table 1. Prolog Rules to Detect IAs for Visitor

```
visitor_pattern(EleList1, EleList2, Hier,CalledMethods1, CalledMethods2):-
        findall(V,(shortname(Method, 'main'), code(Method,Version), nth0(0,Version, V)),
                ListofVersions), nth0(0,ListofVersions, V1),
        object_structure(C,Hier1,ES,V1,ShtMethList1), list_to_set(ES,EleList1),
        nth0(1,ListofVersions,V2),object_structure(C,Hier2,ES1,V2,ShtMethList2),
        list_to_set(ES1,EleList2), called_methods(Hier1,ShtMethList1,EleList1,CM1,V1),
        list_to_set(CM1,CalledMethods1),
        called_methods(Hier2,ShtMethList2,EleList2,CM2,V2),
        list_to_set(CM2,CalledMethods2),subset(CalledMethods1,CalledMethods2),
        subset(EleList1,EleList2), sort(Hier1,Hier),sort(Hier2,Hier).

object_structure(C,Hier,ListofElementClasses, Version, LM):-
        extends(C, 'java.lang.Object'), code(C,CodeInfo),nth0(0,CodeInfo,Version),
        wholeTree(L), flatten(L,Hier), member(C,Hier),
        findall(C1, (((field(F1), context(F1,C), type(F1,C1),code(C1,CodeInfoC1),
                    nth0(0,CodeInfoC1,Version));(localto(Var,_,C,C1))), class(C1),
                    code(C1, Vs), nth0(0,Vs,Version), not(member(C1,Hier))), ES1),
        list_to_set(ES1,ES),
        findall(Ele, (member(Ele,ES), not(member(Ele,Hier))), ListofElementClasses),
                    findall(ShtMethod, (method(Method), code(Method,MethInfo),
                                        nth0(0,MethInfo,Version),
                                        not(shortname(Method,'new')),
                                        once((context(Method,C),
                                        shortname(Method,ShtMethod)))),
                    ListofMethods),
        list_to_set(ListofMethods, LM).

called_methods(Hier,ShtMethList,LE,CalledMethList,Version):-
        findall(CalledMethodsList, (member(Derived,Hier),
                                    not(extends(Derived,'java.lang.Object')),
        findall(Method, (method(M),once(shortname(M, ShtName)),
                        not(ShtName=='new'),once((member(ShtName,ShtMethList),
                        context(M,Derived)))),LM),
                        findall(CalledMethod, (member(M,LM),
                                               callinfo(M,CalledMethod,CallInfo),
                                               nth0(0,CallInfo,Version),
                                               not(shortname(CalledMethod,'new')),
                                               once((context(CalledMethod, EleClass),
                                               code(EleClass,ClassInfo),
                                               nth0(0,ClassInfo,Version),
                                               member(EleClass, LE)))), LC),
        not(LC==[]),list_to_set(LC,CalledMethodsList)), ListofCalledMethodsList),
        list_to_set(ListofCalledMethodsList,List1),nth0(0,List1,List2), length(List2,N),
        flatten(List1,List3),list_to_set(List3,CalledMethList),length(CalledMethList,N).
```

case are values associated with *Hier1, EleList1, EleList2, CalledMethods1 and CalledMethods2* variables.

Once the IAs are identified, these are refactored according to the transformation rules of Visitor pattern. The final refactored code that corresponds to the initial version is shown in Figure 3. This design has features which enables it to be easily extendible.

6 Related Work

Tokuda and Batory [4] proposed a semi automatic approach to refactoring which increases productivity when compared to refactoring by hand. Roberts [5] has proposed a tool that performs composite refactorings in Smalltalk. Mel O Cinneide [6] developed a tool to apply design patterns in refactoring using helper functions and analysis functions. Though these tools use design patterns to refactor, they require manual observation of the code to infer patterns. This is tedious and error-prone.

Our approach is similar to Sang-Uk Jeon et al [13] approach to infer design patterns from historic data (can be multiple versions). In Sang-Uk Jeon et al. approach inference rules are formed based upon human experts experience and knowledge which is accurate to some extent. But, in our approach, inference rules are formed based upon important factors such as, extensive analysis of design patterns and their applicability, metrics [11] to identify (can also be useful in inferring) design patterns from code, program analysis, and mechanic section from Kerievesky catalogue [10]. Further, pre-formed rules makes it easily judge the accuracy and correctness of the inference and they can easily be improved through Artificial Intelligence learning techniques. So, our approach leads to more accurate inference (minimum number of false positives) of design patterns than Sang-Uk Jeon et al approach. Further, our inference rules written in Prolog can also be used (with slight variation) in TyRuBa [14], which is more efficient (faster) than Prolog and it provides lot more flexibility to add , modify, and remove rules as and when required. So, this makes the inference process faster and accurate.

Tourwe and Mens [15] proposed an approach to show how automated support can be provided for identifying refactoring in Smalltalk. However, this approach is not intended for introducing GOF design patterns.

7 Conclusions and Future Work

Some of the patterns can be automatically inferred from a single version of software. Some of the patterns which are meant for extendibility may not be accurately inferred from a single version. Induction of patterns from multiple versions of software makes it more accurate thereby avoiding complexity of the refactored system. This paper presents rules for four design patterns. Changes from version to version are assumed as major changes.

This paper restricts the number of versions to two. The same rules can be adapted to more than two versions with slight variation. As the number of versions increases, so does the accuracy of induction. Currently, the inference rules for four design patterns have been implemented.

References

1. Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.
2. E.Gamma, R.Helm, R.Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software.* Addison Wesley, 1994.
3. William G. Griswold and David Notkin. Automated Assistance for Program Restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.
4. Don Batory and Lance Tokuda. Automated Software Evolution via Design Pattern Transformations. Technical Report CS-TR-95-06, The University of Texas at Austin, Department of Computer Sciences, 1995.
5. Donald Bradley Roberts. *Practical Analysis for Refactoring.* PhD thesis, Univerisity of Illinois at Urbana-Champaign, June 1999.
6. Mel O Cinneide. Automated Refactoring to Introduce Design Patterns . In *International Conference on Software Engineering*, pages 722–724. ACM Press,Limerick , June 2000.
7. J Rajesh and D Janakiram. JIAD: A Tool to Infer Design Patterns in Refactoring. In *Proceedings of 6th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming(PPDP04), Verona, Italy*, pages 227–237, August 24-26, 2004.
8. Wielemaker. Prolog Reference Manual. http://swi.psy.uva.nl/projects/SWI-Prolog.
9. D. Janaki Ram and J. Rajesh. Detecting Intent Aspects from Code to Apply Design Patterns in Refactoring: An Approach Towards a Refactoring Tool. In *Proceedings of 2nd Workshop on Software Design and Architecture(SODA)04*, January 2004.
10. Joshua Kerievsky. *Refactoring to Patterns.* Industrial Logic, Inc, 2002.
11. Taichi Muraki and Motoshi saeki. Metrics for Applying GOF Design Patterns in Refactoring Process. *International WorkShop on Principles of Software Evolution(IWPSE)*, 2001.
12. Isomrophism. http://en.wikipedia.org/wiki/ Isomorphism.
13. Sang-Uk Jeon, Joon-Sang Lee and Doo-Hwan Bae. An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs. In *Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, page 337. IEEE Computer Society Press, 2002.
14. TyRuBa. http://tyruba.sourceforge.net.
15. Tom Tourwe and Tom Mens. Identifying Refactoring Oppurtunities Using Logic Meta Programming. *7th European Conference on Software Maintenance and Reengineering*, page 91, March 2003.